Hardware design for Hash functions

Yong Ki Lee[(1)], Miroslav Knežević[(2)], and Ingrid Verbauwhede[(1),(2)]

[(1)]University of California, Los Angeles, Electrical Engineering,
420 Westwood Plaza, Los Angeles, CA 90095-1594 USA
[(2)]Katholieke Universiteit Leuven, ESAT/COSIC,
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
jfirst@ee.ucla.edu,
{Miroslav.Knezevic,Ingrid.Verbauwhede}@esat.kuleuven.be

Introduction to chapter. Due to its cryptographic and operational key features such as the one-way function property, high speed and a fixed output size independent of input size the hash algorithm is one of the most important cryptographic primitives. A critical drawback of most cryptographic algorithms is the large computational overheads. This is getting more critical since the data amount to process or communicate is dramatically increasing. In many of such cases, a proper use of the hash algorithm effectively reduces the computational overhead. Digital signature algorithm and the message authentication are the most common applications of the hash algorithms. The increasing data size also motivates hardware designers to have a throughput optimal architecture of a given hash algorithm. In this chapter, some popular hash algorithms and their cryptanalysis are briefly introduced, and a design methodology for throughput optimal architectures of MD4-based hash algorithms is described in detail.

Key Words: Hash algorithm, MD-based hash, Crypto-analysis, Throughput optimal architecture, Iteration bound analysis

1 Introduction

Cryptographic hash algorithms are one of the most important primitives in security systems. They are most commonly used for digital signature algorithms [1], message authentication and as a building block for other cryptographic primitives such as hash based block ciphers (Bear, Lion [8] and Shacal [22]), stream ciphers and pseudo-random number generators. Hash algorithms take input strings - M of arbitrary length and translate them to short fixed-length output strings, so called *message digests* - $Hash$(M). The typical example of hash based message authentication is protecting the authenticity of the short hash result instead of protecting the authenticity of the whole message. Similarly, in digital signatures a

signing algorithm is always applied to the hash result rather than to the original message. This ensures both performance and security benefits. Hash algorithms can also be used to compare two values without revealing them. A typical example for this application is a password authentication mechanism [39].

As the hash algorithms are widely used in many security applications, it is very important that they fulfill certain security properties. Those properties can be considered as follows:

1. *Preimage resistance*: It must be hard to find any preimage for a given hash output, *i.e.* given a hash output H getting M must be hard such that H=$Hash$(M).
2. *Second Preimage resistance*: It must be hard to find another preimage for a given input, *i.e.* given M_0 and $Hash(M_0)$ getting M_1 must be hard such that $Hash(M_0)$=$Hash(M_1)$.
3. *Collision resistance*: It must be hard to find two different inputs of the same hash output, *i.e.* getting M_0 and M_1 must be hard such that $Hash(M_0)$=$Hash(M_1)$.

An algorithm that is characterized by the first two properties is called a *One-Way Hash Algorithm*. If all three properties are met the hash algorithm is considered *Collision Resistant*. Finding collisions in a specific hash algorithm is the most common way of attacking it.

There are a few different types of hash algorithms described in literature. They are based on block ciphers, modular arithmetic, cellular automation, knapsack and lattice problem, algebraic matrices, etc. The most commonly used hash algorithms, known as *Dedicated Hash algorithms*, are especially designed for hashing and are not provably secure. The biggest class of these algorithms is based on the design principles of the MD4 family [40].

In this article we show how *retiming* and *unfolding*, well known techniques used in Digital Signal Processing, can be applied as a design methodology for very fast and efficient hardware implementations of MD-based hash algorithms.

2 Popular Hash Algorithms and Their Security Considerations

The design philosophy of the most commonly used hash algorithms such as MD5, SHA family and RIPEMD is based on design principles of the MD4 family. In this section we will give a short overview and provide historical facts about existing attacks on these algorithms.

MD4 is a 128-bit cryptographic hash algorithm introduced by Ron Rivest in 1990 [40]. The MD4 algorithm is an iterative algorithm which is composed of 3 rounds, where each round has 16 hash operations. Therefore, MD4 has 48 iterations. On each iteration (hash operation), intermediate results are produced and used for the next iteration. A hash operation is a combination of arithmetic additions, circular shifts and some Boolean functions. All the operations are based on 32-bit words. The differences among the MD4 families are in the word size, the number of the iterations and the combinations of arithmetic additions and Boolean functions.

After MD4 was proposed several other hash algorithms were constructed based on the same design principles: 256-bit extension of MD4 [40], MD5 [41], HAVAL [50], RIPEMD [3], RIPEMD-160 [17], SHA0 [5], SHA1 [6], SHA-256, SHA-384, SHA-512, SHA-224 [7], etc. The first attack on MD4 was published already in 1991 by den Boer and Bosselaers [14]. The attack was performed on reduced version of MD4 (2 out of 3 rounds). Additionally, in November 1994 Vaudenay shows that the fact that the inert algorithm is not a multipermutation allows to mount an attack where the last round is omitted [43]. In Fall of 1995 Dobbertin finds collisions for all three rounds of MD4 [16]. A few years after Rivest designed the strengthened version MD5 it was shown by den Boer and Bosselaers [9] that the compression function of MD5 is not collision resistant. At the beginning of 1996 Dobbertin also found a free-start collision of MD5 in which the initial value of the hash algorithm is replaced by a non-standard value making the attack possible [15]. Finally, in the rump session of Crypto 2004 it was announced that collisions for MD4, MD5, HAVAL-128 and RIPEMD were found. In 2005 Wang et al. published several cryptanalytic articles [48, 46, 49, 47] showing that use of the differential attack can find a collision in MD5 in less than an hour while the same attack applied to MD4 can be performed in less than a fraction of a second.

The first version of SHA family is known as SHA0 and was introduced by the American National Institute for Standards and Technology (NIST) in 1993 [5]. This standard is also based on the design principles of the MD4 family. One year after proposing SHA0 NIST discovered a certification weakness in the existing algorithm. By introducing a minor change it proposed the new Secure Hash standard known as SHA1 [6]. The message digest size for both algorithms is 160 bits. The first attack on SHA0 was published in 1998 by Chabaud and Joux [10] and was probably similar to the classified attack developed earlier (the attack that yielded

to the upgrade to SHA1). This attack shows that collision in SHA0 can be found after 2^{61} evaluations of the compression function. According to the birthday paradox, a brute force attack would require 2^{80} operations on average. In August 2004 Joux et al. first show a full collision on SHA0 requiring complexity of 2^{51} computations [26]. Finally, in 2005 Wang, Yin and Yu announce full collision in SHA0 in just 2^{39} hash operations [49] and report that collision in SHA1 can be found with complexity of less than 2^{69} computations [47].

The following generation of SHA algorithms known as the SHA2 family was introduced in 2000 and adopted as an ISO standard in 2003 [4]. All three hash algorithms (SHA-256, SHA-384 and SHA-512) have much larger message digest size (256, 384 and 512 bits respectively). The youngest member of this family is SHA-224 and was introduced in 2004 as a Change Notice to FIPS 180-2 [7]. There are only a few security evaluations of the SHA2 algorithms so far. The first security analysis was in 2003 by Gilbert and Handschuh [21] and it showed that neither Chabaud and Joux's attack, nor Dobbertin-style attacks apply to these algorithms. However, they show that slightly simplified versions of the SHA2 family are surprisingly weak. In the same year Hawkes and Rose announce that second preimage attacks on SHA-256 are much easier than expected [23]. Although pointing to the possible weaknesses of SHA2 family these analyses do not lead to actual attacks so far. Cryptanalysis for step-reduced SHA2 can be found in [24].

In [44] Oorschot and Wiener show that in 1994 a brute-force collision search for 128-bit hash algorithm could be done in less than a month with a $10 million investment. Nowadays, according to the Moore's law, the same attack could be performed in less than two hours. As a countermeasure to this attack the size of the hash result has to be at least 160 bits. RIPEMD-160 is hash algorithm with the message digest of 160 bits and was designed by Dobbertin, Bosselaers and Preneel in 1996 [18]. Intention was to make stronger hash algorithm and replace existing 128-bit algorithms such as MD4, MD5 and RIPEMD. To the best of our knowledge the only study concerning security of RIPEMD-160 so far is published by Rijmen et al. [34]. In this analysis the authors extend existing approaches using recent results in cryptanalysis of hash algorithms. They show that methods successfully used to attack SHA1 are not applicable to full RIPEMD-160. Additionally, they use analytical methods to find a collision in 3 rounds variant of RIPEMD-160. To the best of our knowledge no attack has been found for the original RIPEMD-160 algorithm as of 2008.

As a conclusion of this section we would like to point out that hash algorithms such as SHA0, SHA1, MD4 and MD5 are not considered to be secure anymore. As a replacement to these algorithms we would recommend the use of RIPEMD-160 or SHA2 family.

3 Common Techniques Used for Efficient Hardware Implementation of MD4-based Hash Algorithms

Besides the security properties of the hash algorithms, a commonly required property is the high throughput. This becomes more critical since the data amount to process is dramatically increasing every year. Therefore, designing a high throughput architecture for a given hash algorithm is one of the most important issues for the hardware designers.

There are several techniques to increase the throughput in the MD4-based hash algorithm implementations. Due to the same design principles in the MD4-based hash algorithms, most of the techniques used in one algorithm can be used for the others. The most commonly used techniques are pipelining, loop unrolling and using Carry Save Adders (CSA): Pipelining techniques reduce critical path delays by properly positioning registers, whose applications can be found in [12, 13, 32]; Unrolling techniques improve throughput by performing several iterations in a single cycle [35, 33, 11, 31]; CSA techniques reduce arithmetic addition delays of two or more consecutive additions [12, 13, 32, 35, 31]. Many of the published papers combine multiple techniques to achieve a higher throughput. SHA1 is implemented in [35, 31, 36, 20, 42, 45], SHA2 in [12, 13, 33, 11, 31, 42], MD5 in [20, 42, 45, 37, 25] and RIPEMD-160 in [20, 42, 37, 27].

Despite numerous proposals for high throughput hash implementations, a delay bound analysis had been neglected and architecture designs are mostly done by intuition until a recent date. For example, in [12], the authors present a design that achieves the iteration bound, though they do not claim optimality. In fact, their design is the last revision of several other suboptimal attempts [13]. The iteration bound analysis, which define the mathematical upper limit of throughput in the micro-architecture level, of SHA1 and SHA2 are in introduced in [29, 30] recently.

4 Throughput Optimal Architecture of SHA1

The iteration bound analysis starts from drawing a DFG (Data Flow Graph) corresponding to a given algorithm. After analyzing the iteration

bound, we apply transformation techniques such as the retiming transformation and the unfolding transformation, which are comparable to the pipelining and the unrolling respectively. The iteration bound analysis and the transformations are publicly known and proven techniques in the signal processing area [38]. We adopt most of the notation and the definition and formalize a design methodology for MD4-based hash algorithms. A related work can be found in [29, 30].

4.1 The SHA1 Hash Algorithm and its DFG

SHA1 is the most widely used hash algorithm as of 2008. It produces an output of 160 bit length with an arbitrary input size less than 2^{64} bits, and requires 80 iterations to digest one message block of 512 bits. The mathematical expression of SHA1 is described in Fig. 1 where $ROTL^k$ represents a k-bit circular left shift, and K_t is a constant value depending on the number of iterations, t. $M_t^{(i)}$ is the t-th 32-bit word of the i-th message block.

$$F_t(x,y,z) = \begin{cases} (x \wedge y) \vee ((\neg x) \wedge z) & 0 \le t \le 19 \\ x \oplus y \oplus z & 20 \le t \le 39 \\ (x \wedge y) \vee (x \wedge z) \vee (y \wedge z) & 40 \le t \le 59 \\ x \oplus y \oplus z & 60 \le t \le 79 \end{cases}$$

(a) Nonlinear Function

$$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ ROTL^1(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) & 16 \le t \le 79 \end{cases}$$

(b) Expander Computation

$$\begin{aligned} TEMP_t &= ROTL^5(A_t) + F_t(B_t, C_t, D_t) + E_t + W_t + K_t \\ E_{t+1} &= D_t \\ D_{t+1} &= C_t \\ C_{t+1} &= ROTL^{30}(B_t) \\ B_{t+1} &= A_t \\ A_{t+1} &= TEMP_t \end{aligned}$$

(c) Compressor Computation

Fig. 1. SHA1 Hash Computation

In order to perform the iteration bound analysis and the transformation techniques, we need to convert Fig. 1(c) to a DFG. Driving a

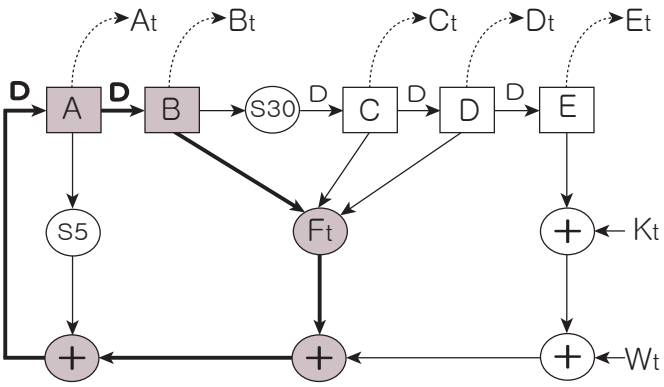


Fig. 2. SHA1 Data Flow Graph

DFG can be done straightforwardly as shown in Fig. 2 where $S5$ and $S30$ represent $ROTL^5$ and $ROTL^{30}$ respectively. The dashed lines are for driving outputs at the t-th iteration and the outputs of the last iteration are used to produce a hash output for one message block. The solid lines indicate the data flow throughout the DFG. Box A, B, C, D and E represent registers, which give the output at cycle t, and circles represent some functional nodes which perform the given functional operations. A D on edges represents an *algorithmic* delay, *i.e.* a delay that cannot be removed from the system. Next to algorithmic delays, nodes also have functional delays. The functional delays are the propagation delays to perform the given operations. We express the functional delays of $+$ and F_t as $Prop(+)$ and $Prop(F_t)$, respectively. The iteration bound analysis assumes that every functional operation is atomic. This means that a functional operation can not be split or merged into some other functional operations. The meaning of the bold lines will be explained in the next sub-section.

4.2 Iteration bound analysis

The iteration bound analysis defines the minimally achievable delay bound of an iterative algorithm and hence it defines the maximally achievable upper bound of throughput. This will not only give designers a goal but also prevent futile efforts to achieve better than the theoretical optimum.

If t_l is the loop calculation time and w_l is the number of algorithmic delays in the l-th loop, the l-th loop bound is defined as t_l/w_l. The iteration bound is the maximum loop bound.

$$T_\infty = \max_{l \in L} \left\{ \frac{t_l}{w_l} \right\} \tag{1}$$

where L is the set of all possible loops.

In Fig. 2, the loop with the maximum loop bound is the one with bold lines and shaded nodes. The shifts of a fixed number are negligible in hardware implementations, and therefore, we ignore the delays of shifts. In Fig. 1(a), it can be seen that the worst case of $Prop(F_t)$ is the "Three Input Bitwise Exclusive OR" operation. This is the same as the critical path of CSA (this fact will be used in the following subsection with more explanation about CSA), and is definitely less than the critical path delay of a 32-bit addition. So, we can assume that $Prop(F_t) \approx Prop(CSA) < Prop(+)$.

Since in the marked loop the loop calculation time is $2 \times Prop(+) + Prop(F_t)$ and the number of algorithmic delays is 2, the iteration bound of the SHA1 hash algorithm can be defined as follows:

$$\begin{aligned} T_\infty^{SHA1} &= \max_{l \in L} \left\{ \frac{t_l}{w_l} \right\} = \frac{2 \times Prop(+) + Prop(F_t)}{2} \\ &= Prop(+) + \frac{Prop(F_t)}{2} \end{aligned} \tag{2}$$

Note that the order of the four adders in SHA1 does not make a difference in the mathematical calculation, which means that there are several different ways to represent a SHA1 DFG. For example, $(a+b)+c$ and $(b+c)+a$ will have different DFGs though they are mathematically equivalent. When a DFG is drawn, the DFG of the minimum iteration bound must be chosen. The DFG in Fig. 2 is one of the DFGs with the minimum iteration bound.

The critical path delay is defined as the maximum calculation delay between any two consecutive algorithmic delays, *i.e.* Ds. In Fig. 2, the critical path delay is $4 \times Prop(+)$, which is larger than the iteration bound. In order to obtain the critical path delay of the iteration bound, we use the retiming and unfolding transformations.

4.3 Iteration bound analysis with Carry Save Adders

In the iteration bound analysis, we assume that each functional node in a DFG can not be merged or split into some other operations. Therefore, in order to use a CSA (Carry Save Adder), we have to draw another DFG.

A CSA produces two values (carry and sum) from three input operands. Since CSA has small area and propagation delay in throughput optimized implementations, it is commonly used when three or more operands are summed. The SHA1 DFG with CSA is drawn in Fig. 3.

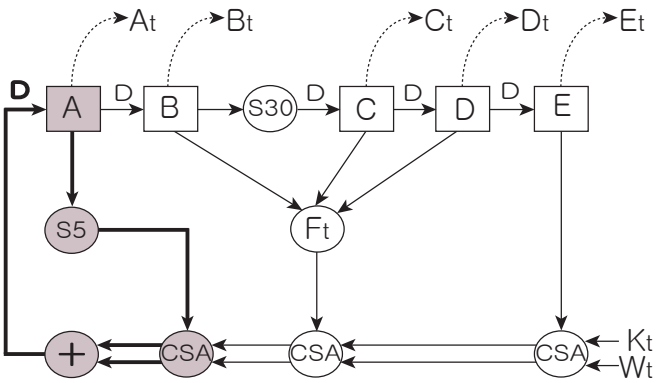


Fig. 3. SHA1 Data Flow Graph with CSA

Since $Prop(F_t) \approx Prop(CSA) < Prop(+)$ as shown in the previous subsection, the loop with the maximum loop bound is the one with bold lines and shaded nodes in Fig. 3 and its iteration bound is given in Eq. (3).

$$T_\infty^{SHA1(CSA)} = Prop(+) + Prop(CSA) \approx Prop(+) + Prop(F_t) \quad (3)$$

Since $T_\infty^{SHA1} < T_\infty^{SHA1(CSA)}$, the use of CSA does not help to reduce the iteration bound. The critical path delay of this case is $Prop(+) + 3 \times Prop(CSA)$ which is also larger than $T_\infty^{SHA1(CSA)}$.

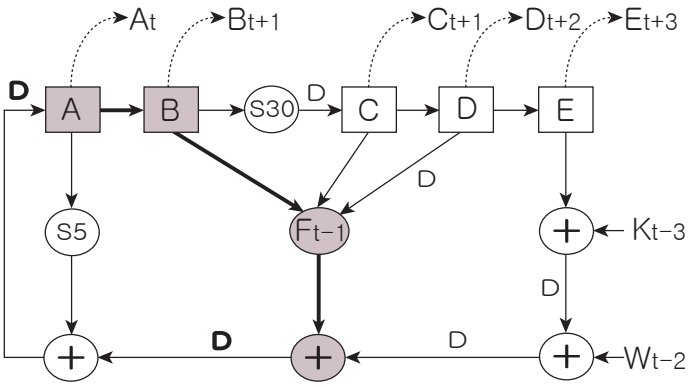
4.4 Retiming Transformation

The attainable minimum critical path delay with the retiming transformation (not using the unfolding transformation) is given in the following Eq. (4) and Eq. (5).

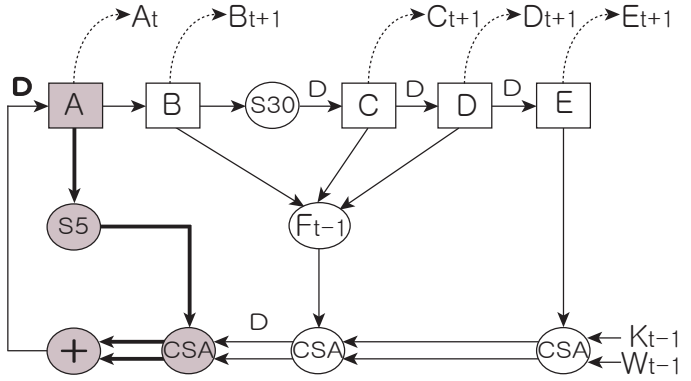
$$\lceil T_\infty^{SHA1} \rceil = \left\lceil \frac{2 \times Prop(+) + Prop(F_t)}{2} \right\rceil = Prop(+) + Prop(F_t) \quad (4)$$

$$\left\lceil T_\infty^{SHA1(CSA)} \right\rceil = Prop(+) + Prop(CSA) \approx Prop(+) + Prop(F_t) \quad (5)$$

Assuming that a functional node can not be split into multiple parts, $\lceil \cdot \rceil$ is the maximum part when the iteration bounds is evenly distributed into N parts, where N is the number of algorithmic delays in a loop, which sits in the denominator. The values of N are denoted by the 2 and 1 in $\lceil T_\infty^{SHA1} \rceil$ and $\left\lceil T_\infty^{SHA1(CSA)} \right\rceil$ respectively. In the case of $\lceil T_\infty^{SHA1} \rceil$ since the iteration bound $2 \times Prop(+) + Prop(F_t)$ can be partitioned into one delay $Prop(+)$ and the other delay $Prop(+) + Prop(F_t)$, the attainable critical path delay by the retiming transformation is $Prop(+) + Prop(F_t)$.



(a) DFG without CSA



(b) DFG with CSA

Fig. 4. Retiming Transformation of SHA1

The retiming transformation modifies a DFG by moving algorithmic delays, *i.e.* Ds, through the functional nodes in the graph. Delays of out-going edges can be replaced with delays from in-coming edges and vice versa. Note that the out-going edges and the in-coming edges must be

dealt as a set independently of the number of out-going or in-coming edges.

Fig. 4 shows the retiming transformation for each of two cases. Even though applying the CSA technique increases the iteration bound, the critical path delays of two cases are similar if there is no unfolding transformation. The time indices of F, K and W are changed due to the retiming transformation. The shaded nodes represent the critical path of each case, which are $A \longrightarrow B \longrightarrow F_{t-1} \longrightarrow +$ and $A \longrightarrow S5 \longrightarrow CSA \longrightarrow +$. Note that there is no propagation delay on A, B and $S5$. Though the two critical paths in Fig. 4 are similar, in practice (b) is preferred since CSA has smaller area than a throughput optimized adder.

Due to the retiming transformation, some of the square nodes are no longer paired with algorithmic delays. Therefore, care must be used to properly initialize the registers and extract the final result: this will be explained in the implementation section (Section 7).

4.5 Unfolding Transformation

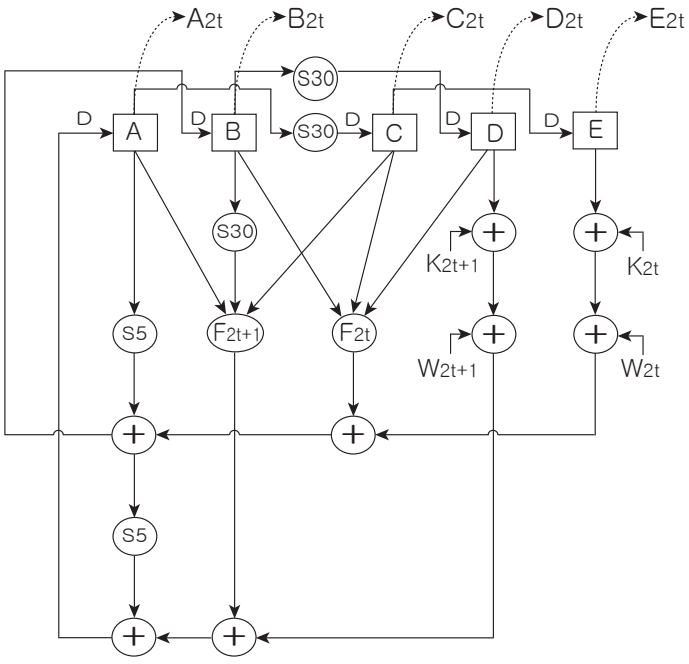


Fig. 5. Unfolding Transformation of SHA1

The DFG of SHA1 with CSA (Fig. 4 (b)) achieves its iteration bound by applying the retiming transformations, but the DFG without CSA (Fig. 4 (a)) does not. Note that since $T_\infty^{SHA1} < T_\infty^{SHA1(CSA)}$, our goal is to achieve T_∞^{SHA1}.

In order to achieve the iteration bound of T_∞^{SHA1}, *i.e.* Fig. 4 (a), we apply the unfolding transformation. The unfolding transformation improves performance by calculating several iterations in a single cycle. The minimally required unfolding factor is the denominator of the iteration bound. This fact can be inferred by noting that the difference between Eq. (2) and Eq. (4) is caused by the un-canceled denominator of the iteration bound. In SHA1, the required unfolding factor is two.

For the unfolding transformation, we expand the equations in Fig. 1(c) to Eq. (6). Now the register values of the time index $t + 2$ are expressed using registers only with the time index t. Note that the functional nodes are doubled due to the unfolding transformation. The resulting DFG is given in Fig. 5. In the indexes of F, K and W, t is also replaced by $2t$

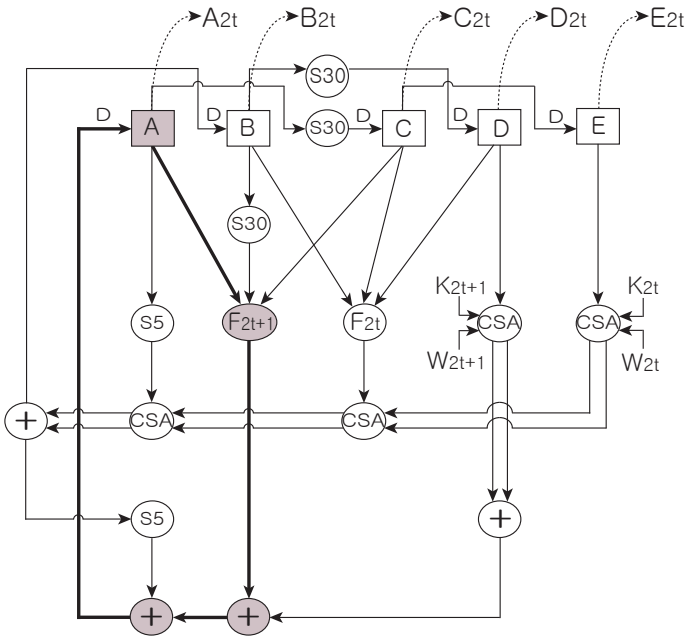


Fig. 6. Unfolding Transformation of SHA1 with CSA

due to the unfolding factor of two.

$$
\begin{aligned}
TEMP_t &= S5(A_t) + F_t(B_t, C_t, D_t) + E_t + W_t + K_t \\
TEMP_{t+1} &= S5(A_{t+1}) + F_{t+1}(B_{t+1}, C_{t+1}, D_{t+1}) + E_{t+1} + W_{t+1} + K_{t+1} \\
&= S5(TEMP_t) + F_{t+1}(A_t, S30(B_t), C_t) + D_t + W_{t+1} + K_{t+1} \\
E_{t+2} &= D_{t+1} = C_t \qquad (6) \\
D_{t+2} &= C_{t+1} = S30(B_t) \\
C_{t+2} &= S30(B_{t+1}) = S30(A_t) \\
B_{t+2} &= A_{A+1} = TEMP_t \\
A_{t+2} &= TEMP_{t+1}
\end{aligned}
$$

After the unfolding transformation, we can substitute two consecutive adders with one CSA and one adder. We have shown in the previous subsection that using CSA does not reduce the iteration bound and therefore does not improve the throughput. However, since CSA occupies less area than a throughput optimized adder, we substitute adders with CSA as long as it does not increase the iteration bound. Fig. 6 shows the DFG which uses CSA. Some consecutive adders are not replaced by CSA since doing so would increase the iteration bound.

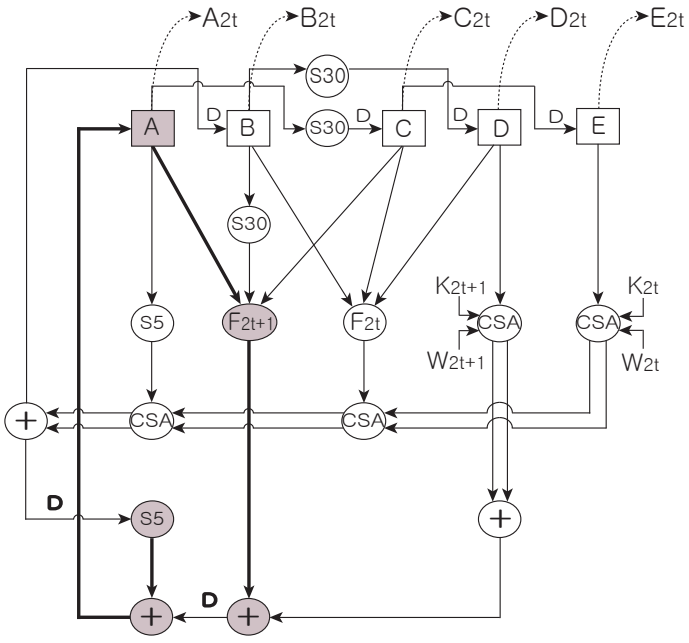


Fig. 7. Unfolding and Retiming Transformation of SHA1

Since $3 \times Prop(CSA) < Prop(+)$, the loop with the maximum loop delay is the loop marked with bold lines in Fig. 6. Finally after performing some proper retiming transformations, we get the result of Fig. 7. The critical path is the path of shaded nodes in Fig. 7 (*i.e.* $S5 \longrightarrow + \longrightarrow A \longrightarrow F_{2t+1} \longrightarrow +$). The normalized critical path delay, $\hat{T}$, can be calculated by dividing the critical path delay by the unfolding factor, which is now equal to the iteration bound as shown in Eq. (7).

$$\hat{T}^{SHA1} = \frac{2 \times Prop(+) + Prop(F_t)}{2} = T_{\infty}^{SHA1} \tag{7}$$

5 Throughput Optimal Architecture of SHA2

$$\begin{aligned}
&Ch(x,y,z) = (x \wedge y) \oplus (\neg x \wedge z)\\
&Maj(x,y,z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)\\
&\Sigma_0^{\{256\}}(x) = ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x)\\
&\Sigma_1^{\{256\}}(x) = ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x)\\
&\sigma_0^{\{256\}}(x) = ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x)\\
&\sigma_1^{\{256\}}(x) = ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)
\end{aligned}$$

(a) SHA-256 Functions

$$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases}$$

(b) SHA-256 Expander Computation

$$\begin{aligned}
&T1_t = H_t + \Sigma_1^{\{256\}}(E_t) + Ch(E_t, F_t, G_t) + K_t^{\{256\}} + W_t\\
&T2_t = \Sigma_0^{\{256\}}(A_t) + Maj(A_t, B_t, C_t)\\
&H_{t+1} = G_t\\
&G_{t+1} = F_t\\
&F_{t+1} = E_t\\
&E_{t+1} = D_t + T1_t\\
&D_{t+1} = C_t\\
&C_{t+1} = B_t\\
&B_{t+1} = A_t\\
&A_{t+1} = T1_t + T2_t
\end{aligned}$$

(c) SHA-256 Compressor Computation

Fig. 8. SHA-256 Hash Computation

The SHA2 family of hash algorithms [7] includes SHA-256, SHA-384 and SHA-512. The input message is expanded into 64 (for SHA-256) or

80 (for SHA-384 or SHA-512) words. The expanded message is again compressed into 256, 384 or 512 bits depending on the algorithm. For one message block, the required iterations are 64 (for SHA-256) or 80 (for SHA-384 or SHA-512). Since all the SHA2 family hash algorithms have the same architecture except for input, output and word sizes, constants, non-linear scrambling functions, *i.e.* Σ_0, Σ_1, Maj, Ch, σ_0 and σ_1, and the number of the iterations, they can be expressed in the same DFG. Therefore, we only consider the SHA-256 algorithm which is shown in Fig. 8.

The design methodology used for SHA1 can be similarly applied to the SHA2 family. However, there are some noticeable differences between the two. The SHA2 family requires only the retiming transformation to achieve their iteration bounds while SHA1 requires both of unfolding and retiming transformations. In the SHA2 family, we also design the expander part since its straightforward architecture has a larger critical path delay than the compressor's iteration bound.

5.1 DFG of SHA2 Compressor

Since within one iteration the order of additions in SHA2 does not affect the results, there are several possible DFGs. A DFG having the minimum iteration bound must be chosen. In SHA2 compressor, since there are only 7 adders, finding a DFG having the minimum iteration bound is not difficult as long as we understand how to calculate the iteration bound.

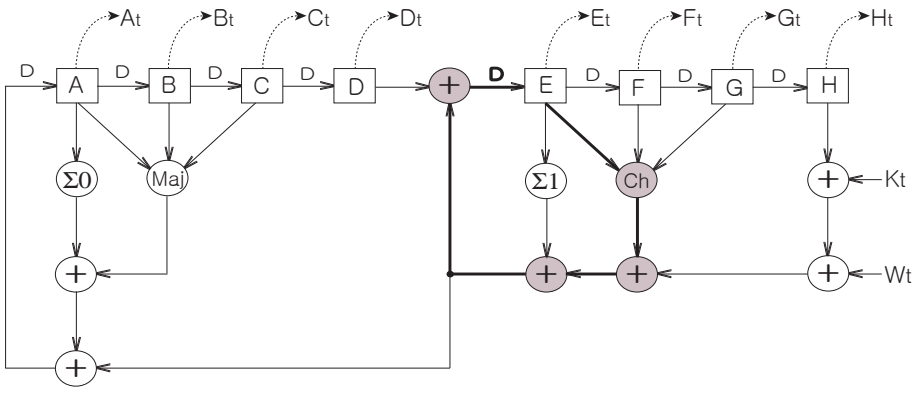


Fig. 9. Basic SHA2 Compressor DFG

The DFG in Fig. 9 is a straightforward DFG. The shaded loop indicates the loop with the largest loop bound and gives the following iteration bound.

$$T_{\infty}^{(9)} = \max_{l \in L} \left\{ \frac{t_l}{w_l} \right\} = 3 \times Prop(+) + Prop(Ch) \tag{8}$$

However, by reordering the sequence of additions, the DFG of Fig. 10 can be obtained which has the smallest iteration bound. As we assume that $Prop(\Sigma 0) \approx Prop(Maj) \approx Prop(\Sigma 1) \approx Prop(Ch)$, the two bolded loops have the same maximum loop bound. Since the loop bound of the left hand side loop cannot be reduced further, no further reduction in the iteration bound is possible. Therefore, the iteration bound of Fig. 10 is as follows.

$$T_{\infty}^{(10)} = \max_{l \in L} \left\{ \frac{t_l}{w_l} \right\} = 2 \times Prop(+) + Prop(Ch) \tag{9}$$

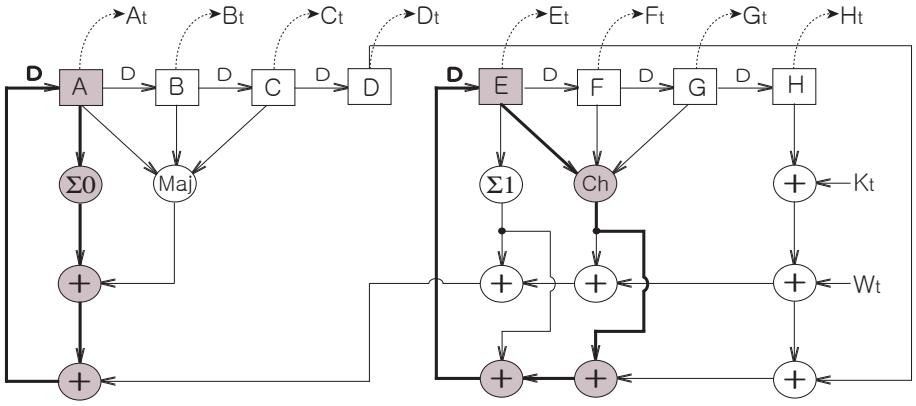


Fig. 10. Optimized SHA2 Compressor DFG

If we assume that any operation in the DFG cannot be merged or split into other operations, the iteration bound of SHA2 is given in Eq. (9). However, if we are allowed to use a Carry Save Adder (CSA), we can substitute two consecutive adders with one CSA and one adder. The resulting DFG is shown in Fig. 11. Note that some of the adders are not replaced with CSA since doing so would increase the iteration bound. Therefore, the final iteration bound is achieved as shown in Eq. (10).

$$T_{\infty}^{SHA2} = \max_{l \in L} \left\{ \frac{t_l}{w_l} \right\} = Prop(+) + Prop(CSA) + Prop(Ch) \tag{10}$$

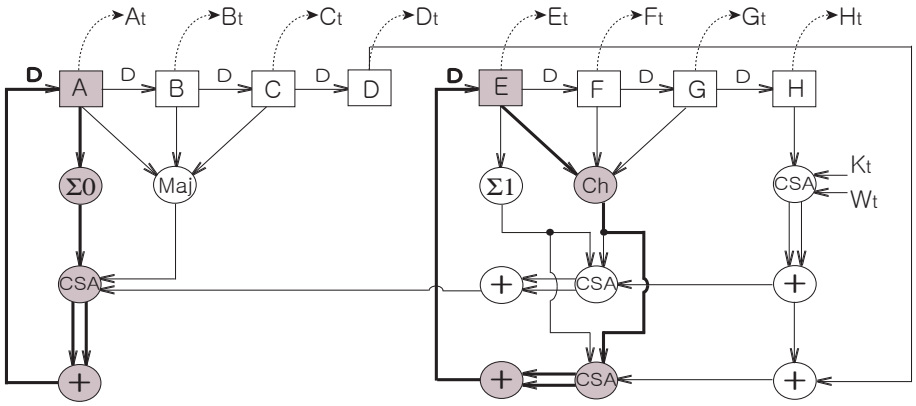


Fig. 11. Optimized SHA2 Compressor DFG with CSA

In the next step, we perform transformations. Since there is no fraction in the iteration bound, we do not need the unfolding transformation. Only the retiming transformation is necessary to achieve the iteration bound. The retimed DFG achieving the iteration bound is depicted in Fig. 12. Note that the indexes of K_{t+2} and W_{t+3} are changed due to the retiming transformation. In order to remove the ROM access time for K_{t+2}, which is a constant value from ROM, we place an algorithmic delay, *i.e.* D, in front of K_{t+2}. This does not change the function.

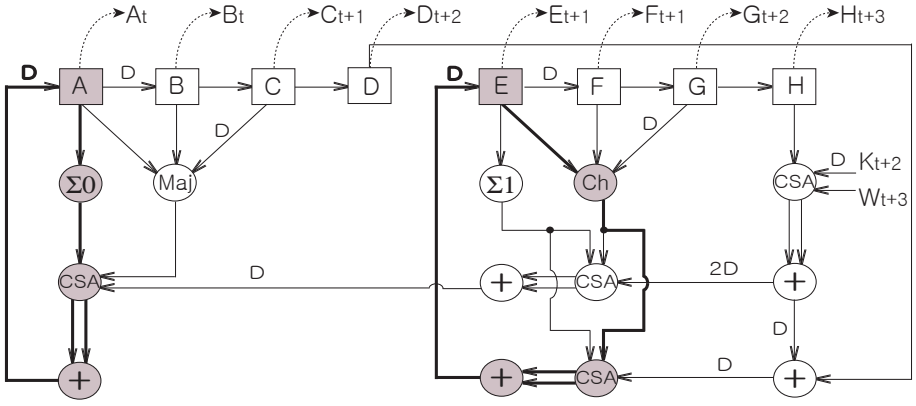
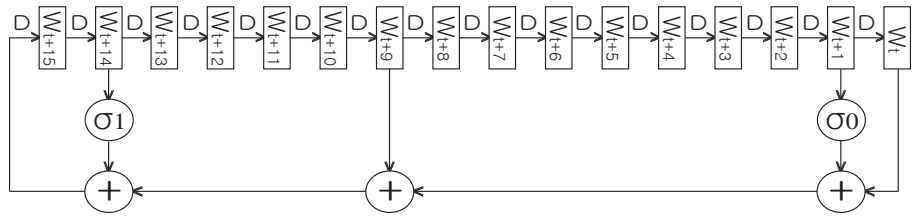
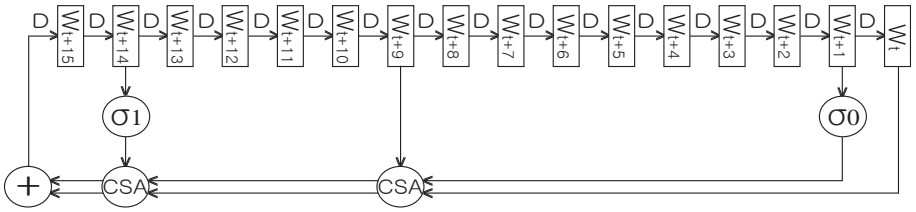


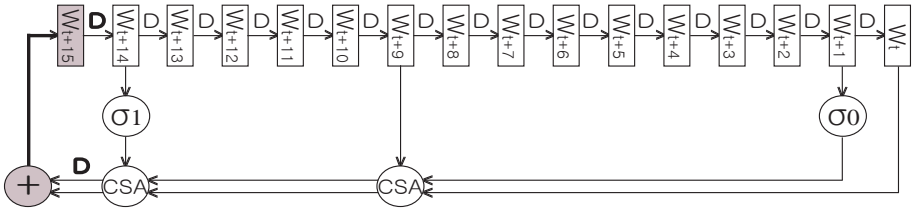
Fig. 12. Final SHA2 Compressor DFG with Retiming Transformation



(a) Basic DFG



(b) DFG with CSA



(c) Retiming Transformed DFG with CSA

Fig. 13. SHA2 Expander DFG

5.2 DFG of SHA2 Expander

A straightforward DFG of the SHA2 expander is given in Fig. 13(a). Even though the iteration bound of the expander is much less than the compressor, we do not need to minimize the expander's critical path delay less than the compressor's iteration bound (the throughput is bounded by the compressor's iteration bound). Fig. 13(b) shows a DFG with CSA, and Fig. 13(c) shows a DFG with the retiming transformation where the critical path delay is $Prop(+)$.

6 Throughput Optimal Architecture of RIPEMD-160

RIPEMD-160 [17] is a hash algorithm designed by Hans Dobbertin et al. in 1996. It is composed of two parallel iterations, where each iteration contains 5 rounds, and each round is composed of 16 hash operations. The

equation and DFG of RIPEMD-160 are shown in Eq. (11) and Fig. 14(a) respectively.

$$
\begin{aligned}
TEMP_t &= S_t\{A_t + F_t(B_t, C_t, D_t) + X_t + K_t\} + E_t \\
E_{t+1} &= D_t \\
D_{t+1} &= S10(C_t) \\
C_{t+1} &= B_t \\
B_{t+1} &= TEMP_t \\
A_{t+1} &= E_t \\
TEMP'_t &= S'_t\left\{A'_t + F'_t(B'_t, C'_t, D'_t) + X'_t + K'_t\right\} + E'_t \\
E'_{t+1} &= D'_t \\
D'_{t+1} &= S10(C'_t) \\
C'_{t+1} &= B'_t \\
B'_{t+1} &= TEMP'_t \\
A'_{t+1} &= E'_t
\end{aligned}
\tag{11}
$$

From Eq. (11) we can see that the two parallel iterations of RIPEMD-160 have identical DFGs. Therefore, we need to analyze only one part and then replicate the results for the second iteration. S_t is a cyclic shift function, X_t is a selection of padded message words and K_t is a constant which depend on the time index t.

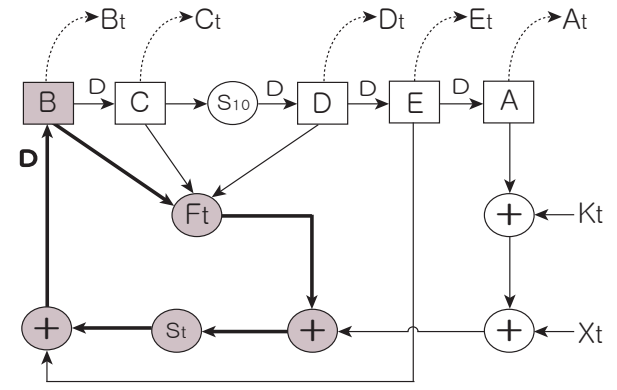
The loop with the maximum loop bound, *i.e.* $B \longrightarrow F_t \longrightarrow + \longrightarrow S_t \longrightarrow + \xrightarrow[D]{} B$, is shown in Fig. 14(a) using shaded nodes, and its iteration bound is shown in Eq. (12).

$$
T_\infty = \max_{l \in L}\left\{\frac{t_l}{w_l}\right\} = 2 \times Prop(+) + Prop(F_t) \tag{12}
$$

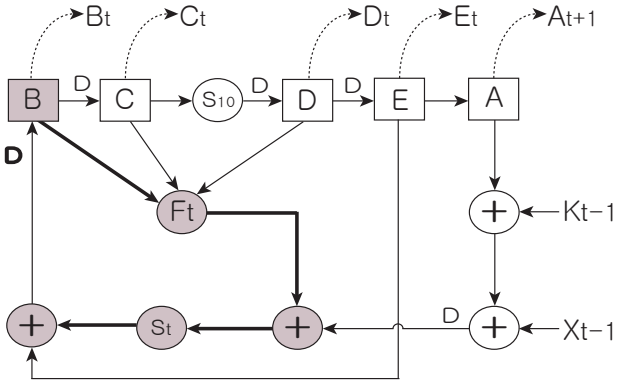
The retiming transformation of RIPEMD-160 which achieves the iteration bound is shown in Fig. 14(b). The critical path is marked by bold line ($B \longrightarrow F_t \longrightarrow + \longrightarrow S_t \longrightarrow +$).

7 Implementation of the designed hash algorithms

In order to verify the design methodology, we synthesized SHA1, SHA2 and RIPEMD-160 using $0.13\mu m$ CMOS standard cell library. We verified that the actual critical paths occur as predicted by our analyses and that the hash outputs are correct.



(a) Data Flow Graph



(b) Retiming Transformation

Fig. 14. RIPEMD-160 Data Flow Graph and its transformation

7.1 Synthesis of the SHA1 Algorithm

For SHA1 we synthesized two versions: one after only the retiming transformation and the other after both the unfolding and retiming transformations. Since the unfolding transformation introduces duplications of functional nodes, its use often incurs a significant increase in area. For the version using only the retiming transformation, we select the DFG in Fig. 4(b) since Fig. 4(b) has less area than Fig. 4(a) with the same critical path delay. Another benefit of Fig. 4(b) is a smaller number of overhead cycles than Fig. 4(a), which will be explained in this section. For the version of the unfolding and retiming transformation, we synthesized the DFG in Fig. 7.

SHA1 with the Retiming Transformation In the transformed DFGs, some of the register values, *i.e.* A, B, ..., E, are no longer paired with an algorithmic delay D. This means that the register B is no longer necessary except for providing the initial value. The retiming transformation moves the delay D associated with register B between two CSAs (we name this delay T) in Fig. 4(b). Though the size of T is doubled (to store both the sum and carry values produced by the CSA), our experiments showed a smaller gate area in Fig. 4(b) than Fig. 4(a) due to the small size of CSA.

Another difference between the original DFG and the transformed DFG occurs during initialization. In the original DFG (Fig. 2), all the registers are initialized in the first cycle according to the SHA1 algorithm. In contrast, initialization requires two cycles in the retimed DFG (Fig. 4(b)). This is because there should be one more cycle to propagate initial values of B, C, D and E into T before the DFG flow starts. In the first cycle, the values of A, B, C, D and E are initialized according to the SHA1 algorithm. At the second cycle, A holds its initial value and C, D, E and T are updated using the previous values of B, C, D and E. From the third cycle, the registers are updated according to the DFG (Fig. 4(b)).

Due to the two cycles of initialization, the retimed DFG introduces one overhead cycle. This fact can also be observed noting that there are two algorithmic delays from E to A. In order to update A with a valid value at the beginning of the iteration, two cycles are required for the propagation. For the case of the retimed SHA1 without CSA (Fig. 4(a)), there are three overhead cycles due to the four algorithmic delays in the path from E to A. Therefore, the required number of cycles for Fig. 4(b) is the number of iterations plus two cycles for initialization, which results in 82 cycles. Since the finalization of SHA1 can be overlapped with the initialization of the next message block, one cycle is excluded from the total number of cycles.

When extracting the final results at the end of the iterations, we should note the indexes of registers. In Fig. 4(b), the index of the output extraction of the register A, *i.e.* A_t, is one less than the others. Therefore, the final result of the register A is available one cycle later than the others.

SHA1 with the Unfolding and Retiming Transformation In the case of Fig. 7, there are 6 algorithmic delays and two of them are not paired with a square node. We name the register for the algorithmic delay between two adders $T1$ and the register for the algorithmic delay between an adder and $S5$ $T2$. However, since $T2$ is equivalent to B, we do

not need a separate register for $T2$. Therefore, the total required registers remain at 5 (retiming does not introduce extra registers in this case).

Since there is only one algorithmic delay in all the paths between any two consecutive square nodes, there is no overhead cycle resulting in the total number of cycles of 41, *i.e.* 40 cycles for iterations plus one cycle for initialization. When extracting the final result of A, the value must be driven from $+(S5(B), T1)$. This calculation can be combined with the finalization since the combined computational delay of $+(S5(B), T1)$, whose delay is $Prop(+)$, and the finalization, whose delay is $Prop(+)$, is $2 \times Prop(+)$ which is less than the critical path delay.

Synthesis Results and Comparison The synthesis results are compared with some previously reported results in Table 1. The 82 cycle version is made by the retiming transformation (Fig. 4(b)), and the 41 cycle version is made by the retiming and the unfolding transformations together (Fig. 7). The throughputs are calculated using the following Eq. (13).

$$Throughput = \frac{Frequency}{\#\ of\ Cycles} \times (512\ bits) \tag{13}$$

Table 1. Synthesis Results and Comparison of SHA1 Hash Algorithm

	Technology (ASIC)	Area (Gates)	Frequency (MHz)	Cycles	Throughput (Mbps)
[36]	0.25μ	20,536	143	82	893
[20]*	0.18μ	70,170	116	80	824.9
[42]	0.13μ	9,859	333.3	85	2,006
[28]	0.18μ	54,133	72.7	12	3,103
[2]	0.18μ	23,000	290	82	1,810
Our Proposal	0.13μ	13,236	943.4	82	5,890
		16,259	558.7	41	6,976

*This is a unified solution for MD5, SHA1 and RIPEMD-160.

In Table 1, the work of [20] is a unified solution for MD5, SHA1 and RIPEMD-160 so its gate count is quite large. The architecture of [28] has a small cycle number and a large gate area due to the unfolding transformation with a large unfolding factor of 8. Even with the use of a large unfolding factor, its critical path delay could not achieve the iteration bound. Comparing our architectures with [42], which is also using 0.13 μm CMOS, ours achieve much higher throughputs.

7.2 Synthesis of the SHA2 Algorithm

In the DFG of Fig. 12, there is no algorithmic delay between registers F and H. Therefore, the values of H will be the same as F except for the first two cycles: in the first cycle, the value of H should be the initialized value of H according to the SHA2 algorithm; in the second cycle the value of H should be the initialized value of G. Therefore, the value of F will be directly used as an input of the following CSA.

Due to the four algorithmic delays from the register H to the register A, there is the overhead of three cycles. Therefore the total number of cycles required for one message block is the number of iterations plus one cycle for initialization and finalization plus three overhead cycles due to the retiming transformation, which results in 68 cycles for SHA256 and 84 cycles for SHA384 and SHA512.

A comparison with other works is shown in Table 2. The throughputs are calculated using the following equation.

$$\begin{aligned} Throughput^{256} &= \frac{Frequency}{\#\ of\ Cycles} \times (512\ bits) \\ Throughput^{384,512} &= \frac{Frequency}{\#\ of\ Cycles} \times (1024\ bits) \end{aligned} \tag{14}$$

Table 2. Synthesis Results and Comparison of SHA2 Family Hash Algorithms

	Algorithm	Technology (ASIC)	Area (Gates)	Frequency (MHz)	Cycles	Throughput (Mbps)
[2]	SHA256	$0.18\mu m$	22,000	200	65	1,575
[42]	SHA256	$0.13\mu m$	15,329	333.3	72	2,370
	SHA384/512		27,297	250.0	88	2,909
[12]	SHA256	$0.13\mu m$	N/A	>1,000	69	>7,420
Our Proposal	SHA256	$0.13\mu m$	22,025	793.6	68	5,975
	SHA384/512		43,330	746.2	84	9,096

Since our HDL programming is done at register transfer level and we have mostly focused on optimizing micro-architecture rather than focusing lower-level optimization, some other reported results, *e.g.* [12], achieve better performance with the same iteration bound delay. However the iteration bound analysis still determines the optimum high level architecture of an algorithm.

7.3 Synthesis of the RIPEMD-160 Algorithm

Using the same design principles, we synthesized RIPEMD-160 algorithm according to Fig. 14(b). Optimizing DFG in this case is rather simple and requires only one retiming transformation. Similarly to the previous implementations the register value A, after retiming transformation, is no longer paired with algorithmic delay D. Therefore, the value of A will be equal to the value of E except for the first cycle. In the first cycle register A is initialized according to the RIPEMD-160 algorithm. For detailed description of initialization step one should refer to [17].

Assuming that the input message is already padded, our implementation of RIPEMD-160 requires 82 clock cycles for calculating the hash result of 512 bit large padded input. One cycle is necessary for initialization, one for finalizing the hash output and 80 cycles for performing each of 5 rounds 16 times. Note here that using additional registers for message expansion is omitted as the padded message can simply be stored in 2 16×32 RAM blocks and appropriate message blocks can be read by providing the correct address values. Storing the message into the RAM cells requires 16 additional cycles and both initialization cycle and hash evaluation step can be performed concurrently with the message storing schedule which makes the total number of cycles equal to 96.

Our result and comparison with previous work is given in Table 3. For synthesis we again use Synopsis Design Vision with $0.13\mu m$ standard cell library. The throughput is calculated according to Eq. (13).

Table 3. Synthesis Results and Comparison of RIPEMD-160 Hash Algorithm

	Technology (ASIC)	Area (Gates)	Frequency (MHz)	Cycles	Throughput (Mbps)
[19]*	0.6μ	10,900 + RAM	59	337	89
[20]**	0.18μ	70,170	116	80	824.9
[42]	0.13μ	24,775	270.3	96	1,442
Our Proposal	0.13μ	18,819 + 2RAM	431	96	2,299

* This is a unified solution for MD5, SHA1 and RIPEMD-160.
**This is a unified solution for MD5, SHA1, SHA-256 and RIPEMD-160.

Observing the results from Table 3 we can conclude that, concerning the speed, our proposal outperforms the previous fastest implementation [42] for almost 60 %. As the size of 2 16×32 RAM blocks is not larger than 10k gates the area of our implementation is comparable to the size of architecture proposed in [42].

8 Hardware Designers' Feedback to Hash Designers

We have shown how a hardware designer can design an architecture of a given MD4-based hash algorithm. As shown in this chapter, the optimal architecture for high throughput is limited by the iteration bound. An improvement of the iteration bound is only possible at the stage of the hash algorithm design. Before concluding this chapter, we would give some feedback to hash algorithm designers for the potential of a better hardware architecture.

Note that the suggestion given in this section is made without considering much of the cryptographic analysis of a hash algorithm. Therefore, it may not be possible to follow the suggestion without sacrificing the security level. We just hope that when a hash designer has some choices, it would be a guideline to choose the one which can result in a better hardware architecture.

8.1 High Throughput Architecture

Minimize the iteration bound by properly placing the algorithmic delays. Achieving a better throughput is directly related with this chapter. If we compare RIPEMD-160 (Fig. 14(a)) with SHA1 (Fig. 2), it is obvious how the maximum throughput can be different depending on the hash algorithm. The iteration bound of RIPEMD-160 is exactly twice of SHA1, *i.e.* RIPEMD-160 would achieve one half throughput of SHA1, though two algorithms look similar regarding their DFGs. This difference is caused by the loops having the iteration bounds. While the marked loop (which determines the iteration bound) of SHA1 has two algorithmic delays, the marked loop of RIPEMD-160 has only one algorithmic delay. Even though the calculation delays of the two loops are the same, one more algorithmic delay of SHA1 resulted in a double throughput of RIPEMD-160. If we modify RIPEMD-160 to have one more algorithmic delay just like SHA1, the throughput of RIPEMD-160 can be doubled. Therefore, the hash designers should consider the placement of the algorithmic delays if the high throughput is one of the design criteria.

8.2 Compact Architecture

Have the iteration bound without a denominator. In SHA1, in order to achieve the maximum throughput, the unfolding transformation was needed. This is caused by the un-canceled denominator of the iteration bound as shown in Eq. (2). The denominator 2 caused the unfolding

transformation with unfolding factor of 2. Note that the unfolding transformation introduces extra circuit area since it duplicates the functional nodes. Therefore, if it is possible to design a hash algorithm without a denominator in the iteration bound, it would not be needed to have the unfolding transformation to achieve the maximum throughput.

Reduce the number of registers. The circuit areas of hash algorithms given in this chapter are dominated by the registers. For example, in our SHA1 implementations (Table 1), the portion of the registers (in 82 cycle version) is about 77% . This situation is similar to the other MD4-based hash algorithms. In this implementation, we use 27 registers of 32-bit words. 11 registers are used in the compressor where 6 are for the variables (*i.e.* to implement the algorithmic delays in Fig. 4(b)), and 5 are for keeping the intermediate results after processing one message block, which is used for the next message block. The other 16 registers are used in the expander. Since the number of registers in the compressor is directly related to the size and the information entropy of the hash output, the reduced number of registers will directly weaken the security level of a hash algorithm. Therefore, the possible way to minimize the registers is in the expander. In the SHA2 expander, for example, shown in Fig. 13, W_t is used to generate W_{t+16}, which has the feedback depth of 16. Therefore, it requires to store one whole message block, *i.e.* 16 words, in the registers. Since there is no much message scrambling activity in the expander compared to the compressor, it may be possible to reduce the feedback depth. For example, if the feedback depth is reduced to 8 from 16, 8 registers can be saved.

Some other comments. Besides those mentioned above, reducing the number of iterations and minimizing the complexity of operations (*i.e.* functional nodes in a DFG) would result in a better hardware architecture and/or throughput. However, doing this must be considered with the possibility of reducing the security level of a hash algorithm. Additionally, we can note that while an addition is a more expensive operation in hardware implementation (since it requires more delay and circuit area) than a non-linear function, a non-linear function is a more expensive operation in software (since it requires more cycles) than an addition.

It must be also noted that a smaller iteration bound does not guarantee a higher throughput architecture, but it just gives a larger upper bound for throughput. If a hash designer wants to have an algorithm to achieve its iteration bound, he/she needs to consider the techniques pre-

sented in this chapter, such as the iteration bound analysis, the unfolding transformation and the retiming transformation.

9 Conclusions and Future Work

In this chapter, we gave a brief overview of MD4-based cryptographic hash algorithms, summarized their security analysis, and introduced a design methodology for the throughput optimal architectures. Among this class of hash algorithms, we chose SHA1, SHA2 and RIPEMD-160 to design and implement for their throughput optimum. Although SHA1 is not considered to be secure anymore, it is still most widely used hash algorithm. For SHA2 and RIPEMD-160 there has been no critical attacks discovered so far, which makes them good candidates for the future cryptographic applications.

Though our implementations shown in this chapter are limited to a few hash algorithms, the design methodology can be applied to any other or new MD4-based hash algorithm. Hash designers may not be familiar with the hardware implementation of an algorithm, so a designed hash algorithm can result in a poor performance. Accordingly, we give some feedback to hash designers.

Concerning the future research, a further optimization in a circuit level could be explored. Note that the presented architectures are claimed to achieve the theoretical optimums in the *micro architecture level*. A lower level optimization, such as designing faster adders or non-linear functions used in hash algorithm, is not considered. Furthermore, it is assumed that the functional nodes of a DFG are *atomic* in the iteration bound analysis. In other words, a given functional node can not be split or merged into some other functional nodes. Therefore, it may be possible to design a faster functional node by merging multiple functional nodes in a given DFG resulting in a smaller iteration bound.

References

1. Digital Signature Standard. In *National Institute of Standards and Technology. Federal Information Processing Standards Publication 186-2.*
2. Helion SHA-1 hashing cores. Helion Technology.
3. RIPE, Integrity Primitives for Secure Information Systems, Final Report of RACE Integrity Primitives Evaluation (RIPE-RACE 1040). LNCS 1007, A. Bosselaers and B. Preneel, Eds., Springer-Verlag, 1995.
4. ISO/IEC 10118-3, Information technology - security techniques - hash functions - Part 3: Dedicated hash functions. 2003.

5. Federal Information Processing Standards Publication 180. Secure Hash Standard. National Institute of Standards and Technology. 1993.
6. Federal Information Processing Standards Publication 180-1. Secure Hash Standard. National Institute of Standards and Technology. 1995.
7. Federal Information Processing Standards Publication 180-2. Secure Hash Standard. National Institute of Standards and Technology. 2003.
8. R. Anderson and E. Biham. Two Practical and Provably Secure Block Ciphers: BEAR and LION. In *International Workshop on Fast Software Encryption (IWFSE'96)*, pages 113–120. LNCS 1039, D. Gollmann, Ed., Springer-Verlag, 1996.
9. B. Boer and A. Bosselaers. Collisions for the Compression Function of MD5. In *Advances in Cryptology, Proceedings of EUROCRYPT'93*, pages 293–304, 1993.
10. F. Chabaud and A. Joux. Differential Collisions in SHA-0. In *Advances in Cryptology, Proceedings of CRYPTO'98*, pages 253–261, 1998.
11. F. Crowe, A. Daly, and W. Marnane. Single-chip FPGA implementation of a cryptographic co-processor. In *Proceedings of the International Conference on Field Programmable Technology (FPT'04)*, pages 279–285, 2004.
12. L. Dadda, M. Macchetti, and J. Owen. An ASIC design for a high speed implementation of the hash function SHA-256 (384, 512). In *ACM Great Lakes Symposium on VLSI*, pages 421–425, 2004.
13. L. Dadda, M. Macchetti, and J. Owen. The design of a high speed ASIC unit for the hash function SHA-256 (384, 512). In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'04)*, pages 70–75, 2004.
14. B. den Boer and A. Bosselaers. An Attack on the Last Two Rounds of MD4. In *Advances in Cryptology, Proceedings of CRYPTO'91*, pages 194–203. LNCS 576, J. Feigenbaum, Ed., Springer-Verlag, 1991.
15. H. Dobbertin. The Status of MD5 After a Recent Attack. In *Cryptographic Laboratories Research*, 1996.
16. H. Dobbertin. Cryptanalysis of MD4. *Journal of Cryptology*, 11:253–271, November 4, 1998.
17. H. Dobbertin, A. Bosselaers, and B. Preneel. RIPEMD-160: A Strengthened Version of RIPEMD. In *Fast Software Encryption*, pages 71–82. LNCS 1039, D. Gollmann, Ed., Springer-Verlag, 1996.
18. H. Dobbertin, A. Bosselaers, and B. Preneel. RIPEMD-160: A Strengthened Version of RIPEMD. In *Fast Software Encryption*, pages 71–82, 1996.
19. S. Dominikus. A Hardware Implementation of MD-4 Family Hash Algorithms. In *Proceedings of the IEEE International Conference of Electronics Circuits and Systems (ICECS'02)*, pages 1143–1146, 2002.
20. T. S. Ganesh and T. S. B. Sudarshan. ASIC Implementation of a Unified Hardware Architecture for Non-Key Based Cryptographic Hash Primitives. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)*, pages 580–585, 2005.
21. H. Gilbert and H. Handschuh. Security Analysis of SHA-256 and Sisters. In *Selected Areas in Cryptography*, pages 175–193, 2004.
22. H. Handschuh and D. Naccache. SHACAL (- Submission to NESSIE -).
23. P. Hawkes, M. Paddon, and G. Rose. On Corrective Patterns for the SHA-2 Family. Cryptology ePrint Archive, Report 2004/207, http://eprint.iacr.org/2004/207, 2004.
24. S. Indesteege, F. Mendel, B. Preneel, and C. Rechberger. Collisions and other Non-Random Properties for Step-Reduced SHA-256. In *Annual Workshop on Selected Areas in Cryptography*. To be appear in LNCS, Springer-Verlag, 2008.

25. K. Järvinen, M. Tommiska, and J. Skyttä. Hardware Implementation Analysis of the MD5 Hash Algorihtm. In *Proceedings of the Annual Hawaii International Conference on System Science (HICSS'05)*, page 298, 2005.
26. A. Joux, P. Carribault, W. Jalby, and C. Lemuet. Collisions in SHA-0. In *Rump session of CRYPTO'04*, 2004.
27. M. Knezevic, K. Sakiyama, Y. K. Lee, and I. Verbauwhede. On the High-Throughput Implementation of RIPEMD-160 Hash Algorithm. In *Proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'08)*, 2008.
28. Y. K. Lee, H. Chan, and I. Verbauwhede. Throughput Optimized SHA-1 Architecture Using Unfolding Transformation. In *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP'06)*, pages 354–359, 2006.
29. Y. K. Lee, H. Chan, and I. Verbauwhede. Iteration Bound Analysis and Throughput Optimum Architecture of SHA-256 (384,512) for Hardware Implementations. In *The 8th International Workshop on Information Security Applications (WISA'07)*, pages 102–114. LNCS 4867, S. Kim, H. Lee, and M. Yung, Eds., Springer-Verlag, 2007.
30. Y. K. Lee, H. Chan, and I. Verbauwhede. Design Methodology for Throughput Optimum Architectures of Hash Algorithms of the MD4-class. *Journal of Signal Processing Systems, Springer, Online first*, 2008.
31. R. Lien, T. Grembowski, and K. Gaj. A 1 Gbit/s partially unrolled architecture of hash functions SHA-1 and SHA-512. In *CT-RSA 2004*, pages 324–338. LNCS 2964, T. Okamoto, Ed., Springer-Verlag, 2004.
32. M. Macchetti and L. Dadda. Quasi-pipelined hash circuits. In *Proceedings of the IEEE Symposium on Computer Arithmetic (ARITH'05)*, pages 222–229, 2005.
33. R. P. McEvoy, F. M. Crowe, C. C. Murphy, and W. P. Marnane. Optimization of the SHA-2 Family of Hah Functions on FPGAs. In *Proceedings of the Emerging VLSI Technologies and Architectures (ISVLSI'06)*, pages 317–322, 2006.
34. F. Mendel, N. Pramstaller, C. Rechberger, and V. Rijmen. On the Collision Resistance of RIPEMD-160. In *Information Security*, pages 101–116, 2006.
35. H. Michail, A.P. Kakarountas, O. Koufopavlou, and C.E. Goutis. A Low-Power and High-Throughput Implementation of the SHA-1 Hash Function. In *IEEE International Symposium on Circuits and Systems (ISCAS'05)*, pages 4086–4089, 2005.
36. Y. Ming-yan, Z. Tong, W. Jin-xiang, and Y. Yi-zheng. An Efficient ASIC Implementation of SHA-1 Engine for TPM. In *IEEE Asia-Pacific Conference on Circuits and Systems*, pages 873–876, 2004.
37. C. Ng, T. Ng, and K. Yip. A Unified Architecture of MD5 and RIPEMD-160 Hash Algorithms. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS'04)*, pages 889–892, 2004.
38. K. K. Parhi. In *VLSI Digital Signal Processing Systems: Design and Implementation*, pages 43–61 and 119–140. Weley, 1999.
39. Bart Preenel. *Encyclopedia of Cryptography and Security, Davies-Meyer Hash Function*. Henk C. A. van Tilborg, Ed., Springer, 2005.
40. R. Rivest. The MD4 message digest agorithm. In *Advances in Cryptology, Proceedings of CRYPTO'90*, pages 303–311. LNCS 537, S. Vanstone, Ed, Springer-Verlag, 1991.
41. R. Rivest. The MD5 Message-Digest Algorithm. Request for Comments: 1321, 1992.

42. A. Satoh and T. Inoue. ASIC-Hardware-Focused Comparison for Hash Functions MD5, RIPEMD-160, and SHS. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'05)*, pages 532–537, 2005.
43. Vaudenay Serge. On the Need for Multipermutations: Cryptanalysis of MD4 and SAFER. In *Fast Software Encryption*, pages 286–297, 1994.
44. Paul C. van Oorschot and Michael J. Wiener. Parallel Collision Search with Cryptanalytic Applications. *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 12(1):1–28, 1999.
45. M. Wang, C. Su, C. Huang, and C. Wu. An HMAC Processor with Integrated SHA-1 and MD5 Algorihtms. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC'04)*, pages 456–458, 2004.
46. X. Wang, X. Lai, D. Feng, H. Chen, and X. Yu. Cryptanalysis of the Hash Functions MD4 and RIPEMD. In *Advances in Cryptology, Proceedings of EUROCRYPT'05*, pages 1–18, 2005.
47. X. Wang, Y. L. Yin, and H. Yu. Finding Collisions in the Full SHA-1. In *Advances in Cryptology, Proceedings of CRYPTO'05*, pages 17–35, 2005.
48. X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In *Advances in Cryptology, Proceedings of EUROCRYPT'05*, pages 19–35, 2005.
49. X. Wang, H. Yu, and Y. L. Yin. Efficient Collision Search Attacks on SHA-0. In *Advances in Cryptology, Proceedings of CRYPTO'05*, pages 1–16, 2005.
50. J. Pieprzyk Y. Zheng and J. Seberry. HAVAL – A one-way hashing algorithm with variable length of output. In *Advances in Cryptology, Proceedings of AUSCRYPT'90*, pages 83–104. LNCS 718, J. Seberry and Y. Zheng, Eds., Spring-Verlag, 1992.